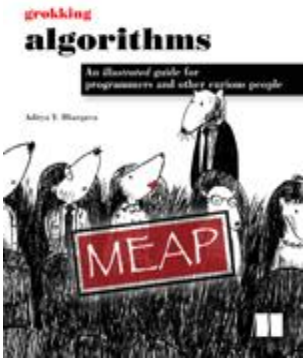


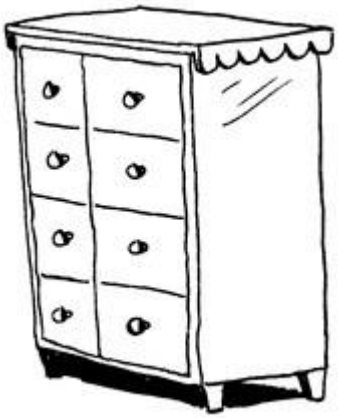
Arrays vs. lists

By Aditya Bhargava, author of Grokking Algorithms

There are two basic ways to store items in memory--arrays and lists. Arrays and lists are used differently depending on your needs. Which is best if you expect to add an item later? What about if you later want to access an item out of order? In this article, based on chapter 2 of Grokking Algorithms, you'll find out which method is more practical for your programming purposes as we explore the advantages and disadvantages of arrays and lists.

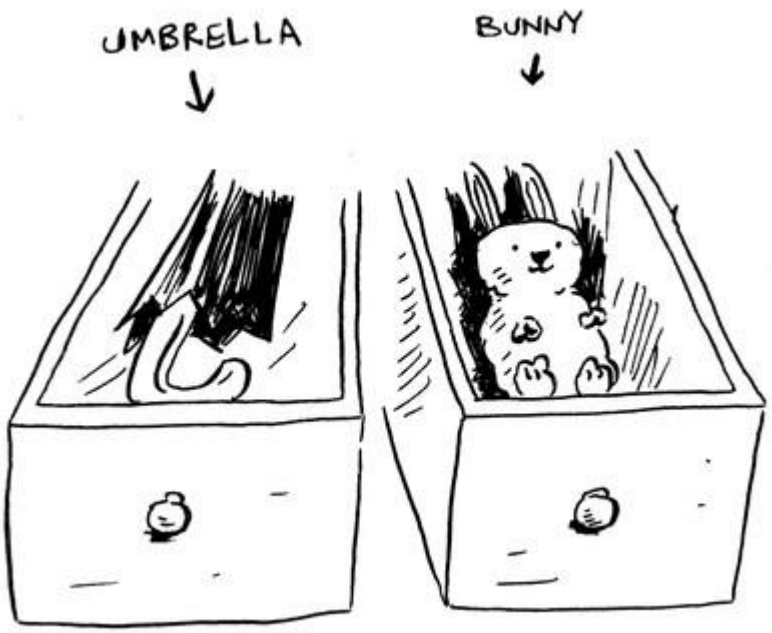
Starting with the basics: How memory works

Imagine you go to a show and need to check your things first. There are a set of drawers available:

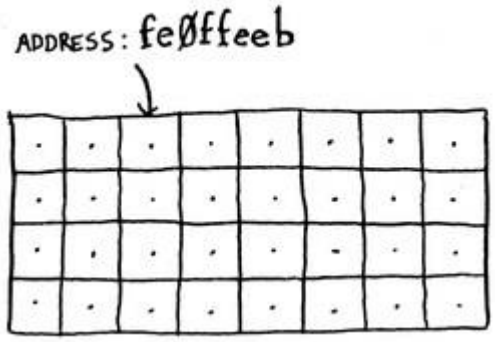


Each drawer can hold one element. You want to store two things, so you ask for two drawers:

You store your two things here:



And you're ready for the show. This is basically how your computer's memory works. Your computer looks like a giant set of drawers, where each drawer has an address:

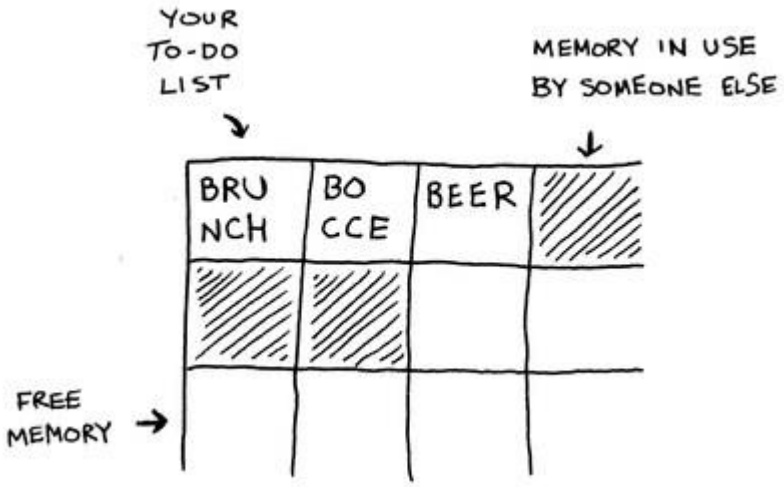


Each time you want to store an item in memory, you ask the computer for some space and it gives you an address where you can store your item. If you want to store multiple items, there are two basic ways to store them: arrays and lists. We'll talk about arrays and lists next, as well as the pros and cons for each one. There isn't one right way to store items for every use case, so it's important to know these differences between arrays and lists.

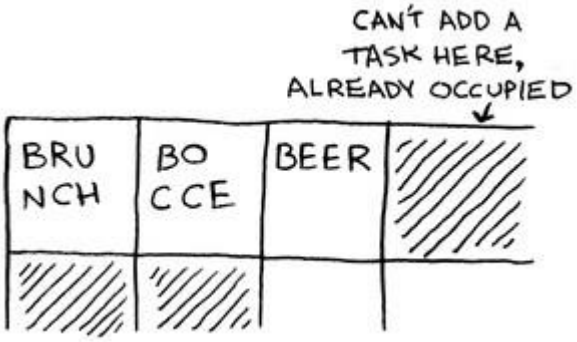
Arrays and linked lists

There are plenty of times when you want to store a list of elements in memory. Suppose you are writing an app to manage your todos. You will want to store your todos as a list in memory:

Should you use an array, or a linked list? Let's store them in an array first, because it is easier to grasp. Using an array means all your tasks are stored contiguously (i.e., right next to each other) in memory:



Now suppose you want to add a fourth task. But the next drawer is taken up by someone else's stuff!



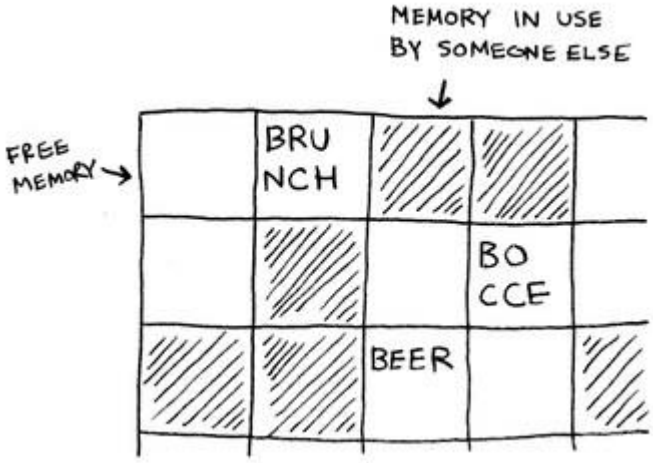
It's like going to a movie with your friends and finding a place to sit…but another friend joins you and there's no place for him. You have to move to a new spot where you all fit. In this case, you need to ask your computer for a different chunk of memory that can fit four tasks. Then you need to move all your tasks there.

Now another friend comes by and again you're out of room. So you all have to move again! What a pain. Similarly, adding new items to your array can be a big pain. If you are out of space every time and need to move to a new spot in memory every time, adding a new item will be really slow. One easy fix is

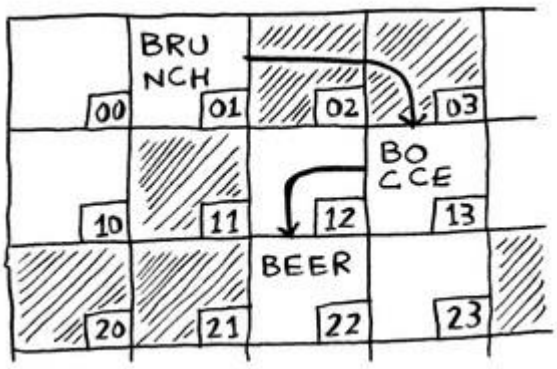
to "hold seats": even if you have just three items in your task list, you might ask the computer for 10 slots just in case. Then, you can add 10 items to your task list without having to move. This is a good workaround, but you should be aware of a couple of downsides:

1. You might not need those extra slots that you asked for, and then that memory is just getting wasted…you aren't using it but no one else can use it either.
2. You might add more than 10 items to your task list and have to move anyway.

So it's a good workaround, but it's not a perfect solution. Linked-lists solve this problem of adding items. With linked lists, your items can be anywhere in memory:



And each item stores the address of the next item in the list. So it's a bunch of random memory addresses "linked" together:



It's like a treasure hunt. You go to the first address, and it says "the next item can be found at address 123." So you go to address 123 and it says "the next item can be found at address 847." And so on. Adding an item to a linked list is easy: you just stick it anywhere in memory, and then store the address with the previous item.

With linked lists, you never have to move your items. You also avoid another problem. Let's say you go to a really popular movie with five of your friends. The six of you are trying to find a place to sit, but the theater is packed. There are no six seats together. Well, sometimes this happens with arrays. Let's say you are trying to find 10,000 slots for your array. Your memory has 10,000 slots, but it doesn't have 10,000 slots together. You can't get space for your array! Well, a linked-list is like saying "let's split up and watch the movie." If there's space in memory, you have space for your linked list.

So if linked-lists are so much better at inserts, what are arrays good for?

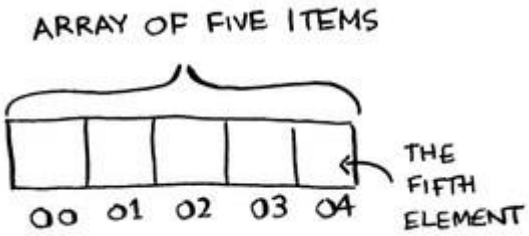
What arrays are good for

Websites with top 10 lists use a really scummy tactic to get more page views. Instead of showing you the list on one page, they put one item on each page and make you click "next" to get to the next item in the list. For example, "Top 10 best TV villains" won't show you the whole list on one page. Instead, you start at #10 (Newman) and have to click next on each page to get to #1 (Gustavo Fring). It gives the websites 10 whole pages on which to show you ads, but it's boring to click "next" nine times to get to #1. It would be much better if the whole list was on one page, and you could click each person's name for more info.

Linked-lists have a similar problem. Suppose you want to read the last item in your linked list. You can't just read it, because you don't know what address it's at. Instead, you have to go to item #1 to get the address for item #2. Then you have to go to item #2 to get the address for item #3. And so on, until you get to the last item. Linked-lists are great if you are going to read all the items one at a time. You can just read one item, and then follow the address to the next item and so on. But if you're going to keep jumping around, linked lists are terrible.

Arrays are different. You know the address for every item in your array. For example, suppose your array is five items, and you know it starts at address 00.

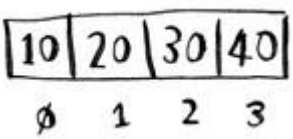
What is the address of item #5?



Just simple math -- it is 04. Arrays are great if you want to read random elements, because you can look up any element in your array instantly. It's like having your top 10 list on a single page.

Terminology

The elements in an array are numbered. This numbering starts from *zero*, not one. For example, in this array, `20` is at position 1:



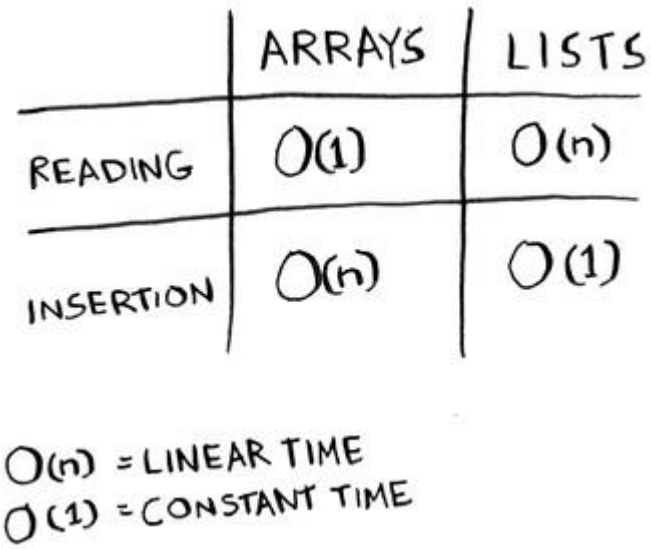
`10` is at position 0. This usually throws new programmers for a loop. Starting at zero makes all kinds of array-based code easier to write, so programmers have stuck with it. Almost every programming language you use will number array elements starting at zero. You will soon get used to it.

The position of an element is called its *index*. So instead of saying "`20` is at position 1," the correct terminology is "`20` is at index 1." I'll be using "index" to mean "position" throughout the rest of this book.

Sidebar: LOCALITY OF REFERENCE

Another advantage to using arrays is that they have something called "locality of reference". Here's what happens: you access the first element of your array. Your computer is smart, and knows that if you access a chunk of memory, you will probably want to access the memory next to it. In this case, if you accessed element #1, you will probably want to access element #2 and #3 too. So it caches that data. Now, accessing the next element in your array will be much faster because it's stored in the cache. Linked-lists don't store their items together, so they don't get this advantage.

So far, you should know that:



EXERCISE

Suppose you are building an app to keep track of your finances:

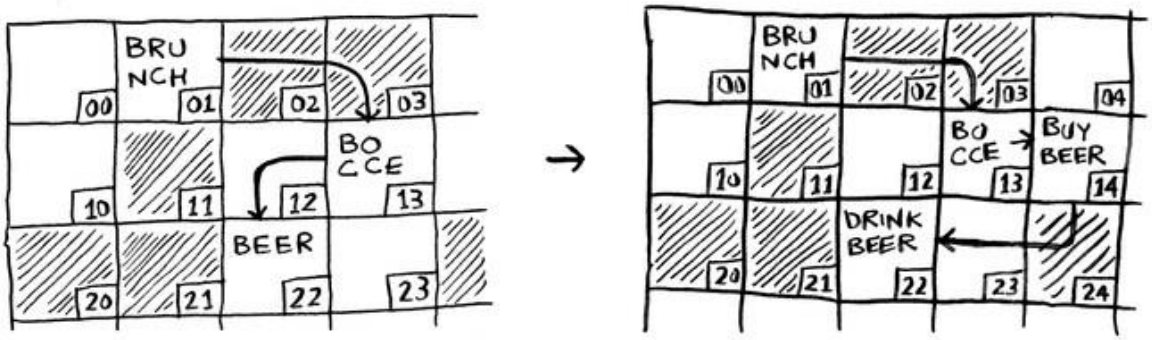
Every day, you write down everything you spent money on. At the end of the month, you review your expenses and sum up how much you spent. Should you use an array or a list?

More insertions and deletions

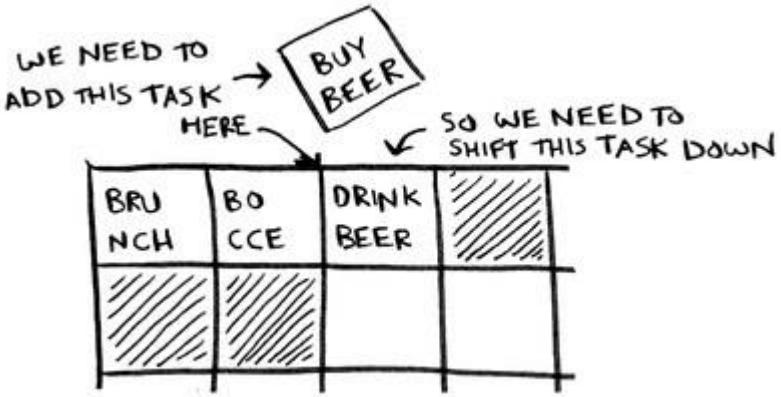
Suppose you want your todo list to work more like a calendar. Earlier, we were adding things to the end of the list:

Now you want to add them in the order they should be done:

What's better if you want to insert elements in the middle: arrays or lists? With lists, it's as easy as changing what the previous element points to:



But for arrays, you have to shift all the rest of your elements down:



And if there's no space, you might have to copy everything to a new location! Lists are better if you want to insert elements into the middle. What if you want to delete an element? Again, lists are better since you just need to change what the previous element points to. With arrays, everything needs to be moved up when you delete an element.

Unlike insertions, deletions will always work. Insertions can fail sometimes when there's no space left in memory. But you can always delete an element.

	ARRAYS	LISTS
READING	O(1)	O(n)
INSERTION	O(n)	O(1)
DELETION	O(n)	O(1)

What gets more use: arrays or lists? Obviously, it depends on the use case. But arrays see a lot of use because usually you need to read things more often than you need to insert things. Arrays and lists are used to implement other data structures too.

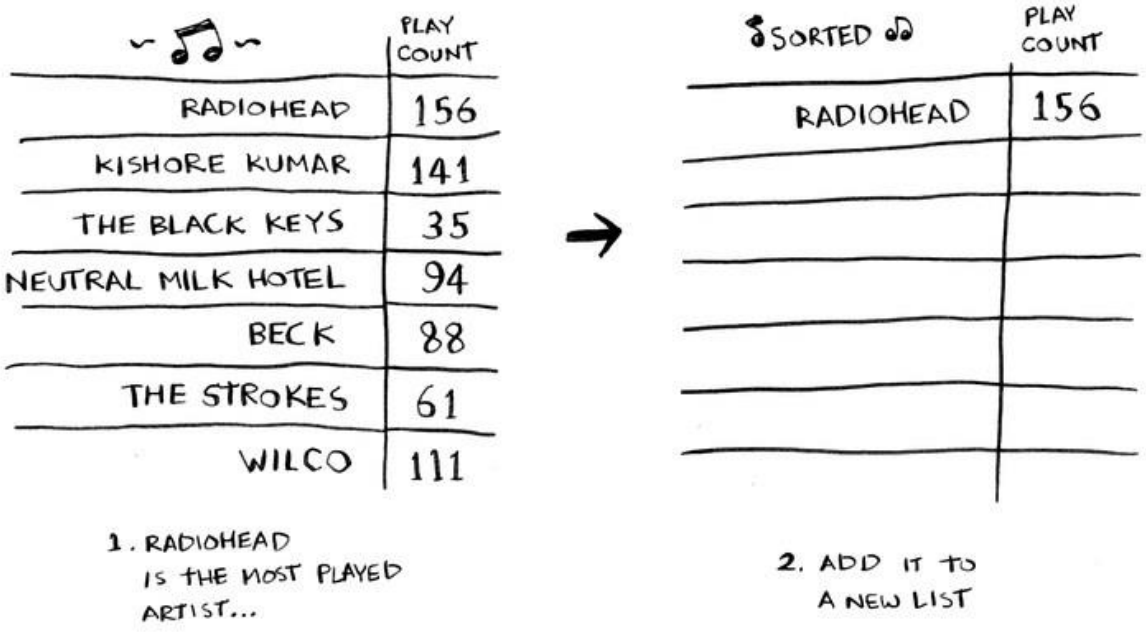
Selection sort

Let's put it all together to learn your second algorithm: selection sort. To follow this section, you will need to understand arrays and lists.

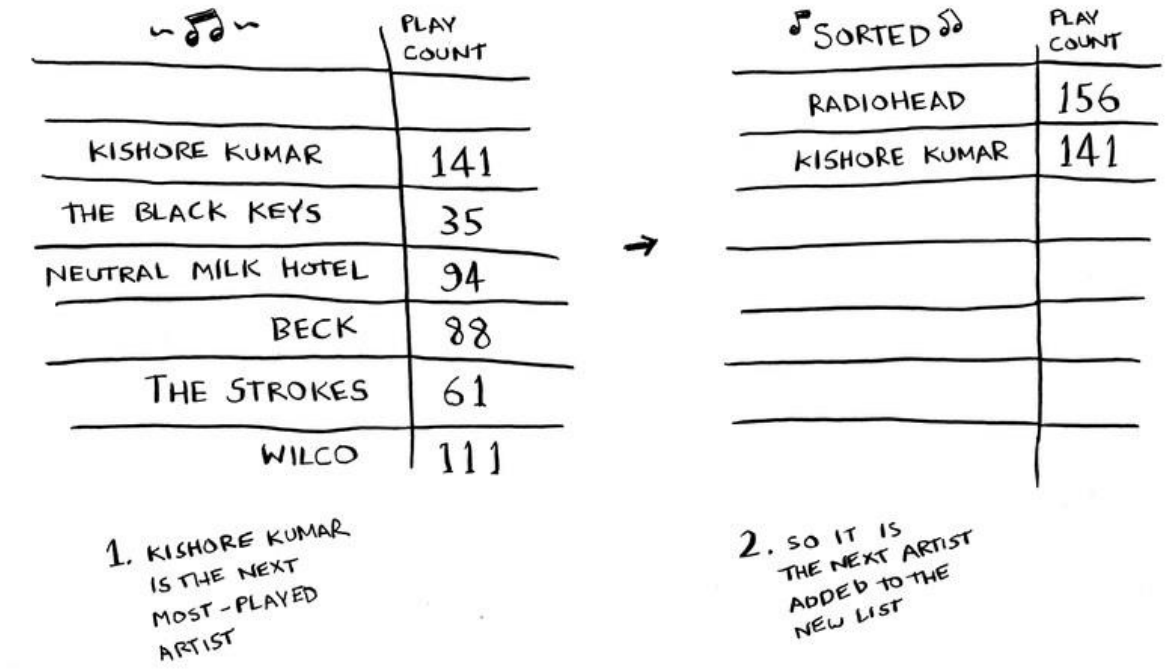
Suppose you have a bunch of music on your computer. For each artist, you have a play count:

	PLAY COUNT
RADIOHEAD	156
KISHORE KUMAR	141
THE BLACK KEYS	35
NEUTRAL MILK HOTEL	94
BECK	88
THE STROKES	61
WILCO	111

You want to sort this list from most- to least-played, so that you can rank your favorite artists. How would you do it? One way is to go through the list, and find the most played artist. Add that artist to a new list:



Do it again to find the next most-played artist:



Keep doing this, and you will end up with a sorted list:

Let's put on our computer science hats and see how long this will take to run. To find the artist with the highest play count, you have to check each item in the list. This takes O(n) time. And you have to do it for each of the elements. And you have to do this for each element in the array. So the whole algorithm runs in

O(n*n) or $O(n^2)$ time.

Congratulations, you just learned your first sorting algorithm!

Sorting algorithms are very useful. Now you can sort:

- names in a phone book travel
- dates
- emails (newest to oldest)

This article is excerpted from the book Grokking Algorithms by Aditya Bhargava. *Grokking Algorithms* is a disarming take on a core computer science topic in which you'll learn how to apply common algorithms to the practical problems you face in day-to-day life as a programmer. You'll start with problems like sorting and searching. As you build up your skills in thinking algorithmically, you'll tackle more complex concerns such as data compression or artificial intelligence. Whether you're writing business software, video games, mobile apps, or system utilities, you'll learn algorithmic techniques for solving problems that you thought were out of your grasp. By the end of the book, you will know some of the most widely applicable algorithms as well as how and when to use them.